Orio: An Annotation-Based Empirical Performance Tuning Framework

Overview

Orio is an extensible annotation system, implemented in Python, that aims to improve both performance and productivity by enabling software developers to insert annotations into their source code (in C or Fortran) that trigger a number of low-level performance optimizations on a specified code fragment. The tool generates many tuned versions of the same operation using different optimization parameters, and performs an empirical search for selecting the best among multiple optimized code variants.

Download

Orio 0.2.1 (alpha) (Open-source license)

Development version (unstable) can be checked out anonymously with:

```
svn co https://svn.mcs.anl.gov/repos/performance/orio
```

Contact Boyana Norris if you wish to contribute to Orio development.

Software Requirements

The requirement of instaling and using Orio is Python, which is widely available in any Linux/Unix distribution. Orio has been tested successfully with Python 2.5 and 2.6 on various Linux distributions, Blue Gene/P and Mac OS X 10.4 and 10.5.

Quick Install

The Orio installation follows the standard Python Module Distribution Utilities, or Disutils for short.

For users who want to quickly install Orio to the standard locations of third-party Python modules (requiring superuser privileges in a Unix system), the installation is straightforward as shown below.

```
% tar -xvzf orio-X.X.X.tar.gz
% cd orio-X.X.X
% python setup.py install
```

On a Unix platform, the above `install` command will normally put an `orcc` script in the `/usr/bin` location, and also create an `orio` module directory in the `/usr/lib/pythonX.X/site-packages` location.

To test whether Orio has been properly installed in your system, try to execute `orcc` command as given below as an example.

```
% orcc --help

description: compile shell for Orio

usage: orcc [options] <ifile>
  <ifile>   input file containing the annotated code

options:
  -h, --help                      display this message
  -o <file>, --output=<file>      place the output to <file>
  -v, --verbose                   verbosely show details of the results of the running program
```

In order to install Orio to an alternate location, users need to supply a base directory for the installation. For instance, the following command will install an `orcc` script under `/home/username/bin`, and also put an `orio` module under `/home/username/lib/pythonX.X/site-packages`. The `orf` script can be used to generate Fortran code (note that Fortran support is currently under development and is thus limited).

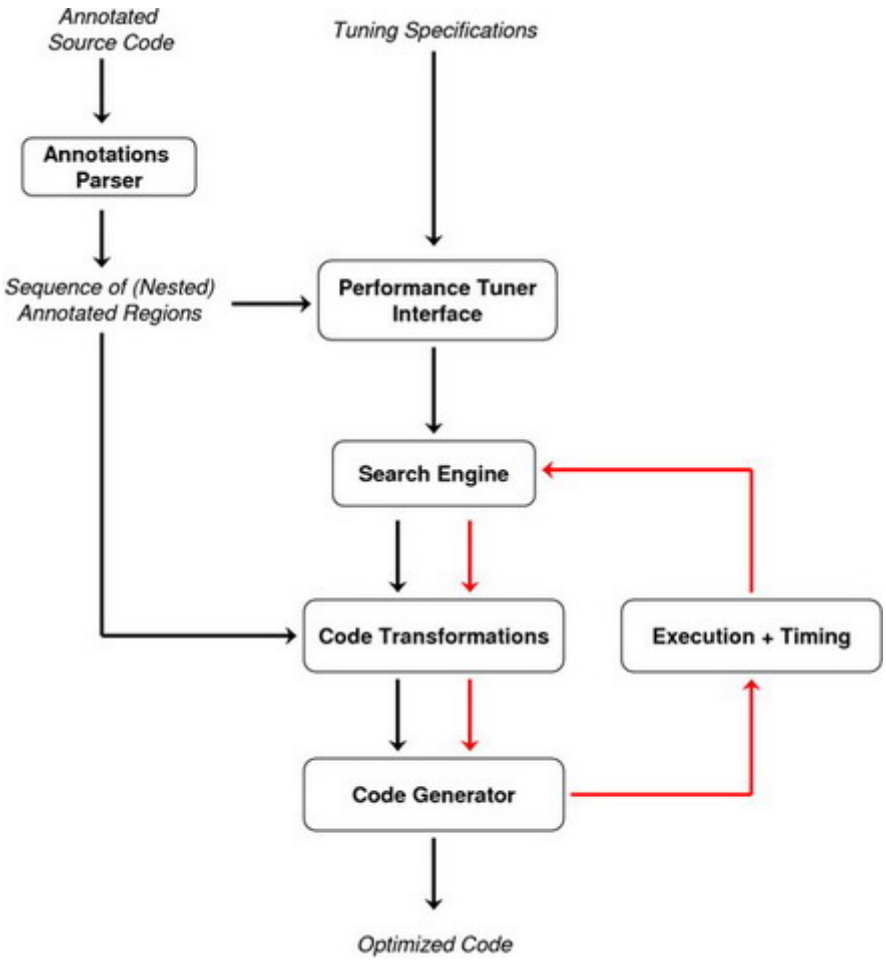
```
% tar -xvzf orio-X.X.X.tar.gz
% cd orio-X.X.X
% python setup.py install --prefix=/home/username
```

It is also important to ensure that the installed Orio module location is included in the PYTHONPATH environment variable. Similarly, users can optionally include the installed `orcc` script location in the PATH shell variable. To do this for the above example, the following two lines can be added in the `.bashrc` configuration file (assuming the user uses Bash shell, of course).

```
export PYTHONPATH=$PYTHONPATH:/home/username/lib/pythonX.X/site-packages
export PATH=$PATH:/home/username/bin
```

Structure of Orio Framework

The picture shown below depicts at a high level the structure and the optimization process of the Orio framework.



As the simplest case, Orio can be used to speed up code performance by performing a *source-to-source transformation* such as loop unrolling and memory alignment optimizations. First, Orio takes a C code as input, which contains syntactically structured comments utilized to express various performance-tuning directives. Orio scans the annotated input code and extracts all annotation regions. Each annotation region is then passed to the code transformation modules for potential optimizations. Next, the code generator produces the final code with various optimizations being applied.

Furthermore, Orio can also be used as an *automatic performance tuning* tool. The system uses its code transformation modules and code generator to generate an optimized code version for each distinct combination of performance parameters. And then, the optimized code version is empirically executed and measured for its performance, which is subsequently compared to the performances of other previously tested code variants. After iteratively evaluating all code variants under consideration, the best-performing code is generated as the final output of Orio.

Because the space of all possible optimized code versions can be exponentially large, an exhaustive exploration of the search space becomes impractical. Therefore, several search heuristics are implemented in the search engine component to find a code variant with near-optimal performance.

The tuning specifications, written by users in the form of annotations, are parsed and used by Orio to guide the search and tuning process. These specifications include important information such as the used compilers, the search strategy, the program transformation parameters, the input data sizes, and so on.

User Guide

As previously discussed , Orio has two main functions: a *source-to-source transformation tool* and an *automatic performance tuning tool*. In the following subsections, simple examples are provided to offer users the quickest way to begin using Orio. But first, a brief introduction to the annotation language syntax is presented next.

Annotation Language Syntax

Orio annotation is denoted as a stylized C comment that starts with '`/*@`' and ends with `@*/`. For instance, the annotation `/*@ end @*/` is used to indicate the end of an annotated code region. The following simple grammar illustrates the fundamental structure of Orio annotations.

> <annotation-region> ::= <leader-annotation> <annotation-body> <trailer-annotation>
> <leader-annotation> ::= `/*@ begin` <module-name> `(` <module-body> `) @*/`
> <trailer-annotation> ::= `/*@ end @*/`

An *annotation region* consists of three main parts: *leader annotation*, *annotation body*, and *trailer annotation*. The annotation body can either be empty or contain C code that may include other nested annotation regions. A leader annotation contains the *module name* of the code transformation component that is loaded dynamically by Orio. A high level abstraction of the computation and the performance hints are coded in the *module body* inside the leader annotation and are used as input by the transformation module during the transformation and code generation phases. A trailer annotation, which has a fixed form (i.e. `/*@ end @*/`), closes an annotation region.

A concrete example of an annotated application code can be seen in the next subsection.

Using Orio as a Source-to-Source Code Transformation Tool

Orio has several code transformation module that have already been implemented and are ready to use. One of the transformation modules is *loop unrolling*, which is a loop optimization that aims to increase register reuse and to reduce branching instructions by combining instructions that are executed in multiple loop iterations into a single iteration. The below sample code demonstrates how to annotate an application code with a simple portable loop unrolling optimization, where the unroll factor used in this example is four. The original code to be optimized in this example is commonly known as AXPY-4, which is an extended version of the AXPY Basic Liner Algebra Subprogram.

```
/*@ begin Loop (
    transform Unroll(ufactor=4)
    for (i=0; i<=N-1; i++)
      y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
) @*/
for (i=0; i<=N-1; i++)
   y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
/*@ end @*/
```

In order to apply loop unrolling to the above code, run the following Orio command (assuming that the annotated code is stored in the file `axpy4.c`).

```
% orcc axpy4.c
```

By default, the transformed output code is written to the file `_axpy4.c`. Optionally, users can specify the name of the output file using the command option `-o <file>`. Below is how the output code looks like.

```
/*@ begin Loop (
    transform Unroll(ufactor=4)
    for (i=0; i<=N-1; i++)
      y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
) @*/
#if ORIGCODE
  for (i=0; i<=N-1; i++)
    y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
#else
  for (i=0; i<=N-4; i=i+4) {
    y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
    y[i+1] = y[i+1] + a1*x1[i+1] + a2*x2[i+1] + a3*x3[i+1] + a4*x4[i+1];
    y[i+2] = y[i+2] + a1*x1[i+2] + a2*x2[i+2] + a3*x3[i+2] + a4*x4[i+2];
    y[i+3] = y[i+3] + a1*x1[i+3] + a2*x2[i+3] + a3*x3[i+3] + a4*x4[i+3];
  }
  for (; i<=N-1; i=i+1)
    y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
#endif
/*@ end @*/
```

In this AXPY-4 example, the name of the code transformation module used to perform loop unrolling is `Loop`. The AXPY-4 computation is rewritten in the module body along with the loop unrolling performance hints (i.e. an unroll factor of four). The resulting unrolled code comprises two loops: one loop with the fully unrolled body, and another loop for any remaining iterations that are not executed in the unrolled loop. Additionally, the generated code include the original code (initially written in the annotation body area) that can be executed through setting the ORIGCODE preprocessor variable accordingly.

More examples on using Orio's source-to-source transformation modules are available in the `orio/testsuite` directory, which can also be browsed online here.

Using Orio as an Automatic Performance Tool

To enhance the performance of a program on target architecture, most compilers select the optimal values of program transformation parameters using analytical models. In contrast, Orio adaptively generates a large number of code candidates with different parameter values for a given computation, followed by empirical executions of these code variants on the target machine. Then the code that yields the best performance is chosen. Orio automates such empirical performance tuning process using annotations, as exemplified in the following simple program.

```
/*@ begin PerfTuning (
 def build {
   arg build_command = 'gcc -O3';
 }
 def performance_params {
   param UF[] = range(1,33);
 }
 def input_params {
   param N[] = [1000,10000000];
 }
 def input_vars {
   decl static double y[N] = 0;
   decl double a1 = random;
```

```
   decl double a2 = random;
   decl double a3 = random;
   decl double a4 = random;
   decl static double x1[N] = random;
   decl static double x2[N] = random;
   decl static double x3[N] = random;
   decl static double x4[N] = random;
 }
) @*/
int i;
/*@ begin Loop (
    transform Unroll(ufactor=UF)
    for (i=0; i<=N-1; i++)
      y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
) @*/
for (i=0; i<=N-1; i++)
  y[i] = y[i] + a1*x1[i] + a2*x2[i] + a3*x3[i] + a4*x4[i];
/*@ end @*/
/*@ end @*/
```

The tuned application in the given example is the same AXPY-4 used in the earlier subsection. The goal of the tuning process is to determine the most optimal value of the unroll factor parameter for different problem sizes. The code located in the `PerfTuning` module body section defines the *tuning specifications* that include the following four basic definitions:

- *build*: to specify all information needed for compiling and executing the optimized code
- *performance_params*: to specify values of parameters used in the program transformations
- *input_params*: to specify sizes of the input problem
- *input_vars*: to specify both the declarations and the initializations of the input variables

So in this example, the transformed AXPY-4 code is compiled using GCC compiler with the -O3 option to activate all its optimizations. The unroll factor values under consideration extends over integers from 1 to 32, inclusively. The AXPY-4 computation is tuned for two distinct problem sizes: N=1K and N=10M. Also, all scalars and arrays involved in the computation are declared and initialized in the tuning specifications to enable the performance testing driver to empirically execute the optimized code.

As discussed before, Orio performance tuning is performed for each different problem size. The number of generated programs is therefore equivalent to the number of distinct combinations of input problem sizes. So, there are two generated program outputs in the AXPY-4 example. Using the default file naming convention, `_axpy_N_1000.c` and `_axpy_N_10000000.c` output files represent the outcomes of Orio optimization process for input sizes N=1K and N=10M, respectively.

See this documentation for more details about the Orio's performance tuning specifications.

Selecting Parameter Space Exploration Strategy

A conceptually straightforward approach to exploring the space of the parameter values is via an exhaustive search procedure. However, this exhaustive approach often becomes infeasible because the size of the search space can be exponentially large. Hence, a proper search heuristic becomes a critical component of an empirical tuning system. In addition to an *exhaustive search* and a *random search*, two effective and practical search heuristic strategies have been developed and integrated into the Orio?s search engine. These heuristics include the *Nelder-Mead Simplex* method and *Simulated Annealing* method. The exhaustive approach is selected as the default space exploration method of Orio; however, Orio user can indicate his preferred search strategy in the tuning specifications, for

instance, using the following *search* definition.

```
def search {
 arg algorithm = 'Simplex';
 arg time_limit = 10;
 arg total_runs = 10;
 arg simplex_local_distance = 2;
 arg simplex_reflection_coef = 1.5;
 arg simplex_expansion_coef = 2.5;
 arg simplex_contraction_coef = 0.6;
 arg simplex_shrinkage_coef = 0.7;
}
```

Orio users can also specify the terminating criteria of the search strategies by providing values to the arguments *time_limit* and *total_runs*. If the search time exceeds the specified time limit, the search is suspended and then Orio returns the best optimized code so far. The total number of runs enforces the search to finish in a specific quantity of *full* search moves. So, the example above indicates that the Simplex search method must terminate within ten-minute time constraint and within ten search convergences.

A search technique sometimes has several parameters that need to be specified. For instance, the Nelder-Mead Simplex algorithm necessitates four kinds of coefficients: *reflection*, *expansion*, *contraction*, and *shrinkage*; and all of these coefficients have default values already defined in the Orio implementation. To alter the values of these algorithm-specific parameters, users can optionally specify them in the tuning specifications. In the example presented above, all arguments with names that start with `simplex_` are called search parameters specifically designed to steer the Simplex algorithm.

To further improve the quality of the search result, each search heuristic is enhanced by applying a local search after the search completes. The local search compares the best performance with neighboring coordinates. If a better coordinate is discovered, the local search continues recursively until no further improvement is possible. In the previous example, users can adjust the distance of the local search by modifying the value of the argument *simplex_local_distance*. A local distance of two implies that the local search examines the performances of all neighbors within a distance of two. It is important to note that the local search is turned off by default for all search heuristics. Thus to activate the local search, Orio users must explicitly assign a positive integer value to the *local_distance* algorithm-specific argument.

The following table lists information about the search techniques implemented in the Orio's search engine.

Search technique	**Keyword**	**Algorithm-specific argument**	**Default value**	**Argument description**
Exhaustive	`'Exhaustive'`	-	-	-
Random	`'Random'`	`local_distance`	0	* the maximum distance of neighboring coordinates considered by the local search

Nelder-Mead simplex	`'Simplex'`	`local_distance`
		`reflection_coef`
		`expansion_coef`
		`contraction_coef`
		`shrinkage_coef`
			0
			1.0
			2.0
			0.5
			0.5
				* the maximum distance of neighboring coordinates considered by the local search
				* the amplitude/intensity of the reflection move
				* the amplitude/intensity of the expansion move
				* the amplitude/intensity of the

				contraction move * the amplitude/intensity of the shrinkage move
Simulated annealing	'Annealing'	local_distance	0	* the maximum distance of neighboring coordinates considered by the local search
		cooling_factor	0.95	* the temperature reduction factor
		final_temperature_ratio	0.05	* the percentage of the termination temperature
		trials_limit	100	* the maximum limit of numbers of search trials at each temperature
		moves_limit	20	* the maximum limit of numbers of successful search moves at each temperature

Developer Guide

This section covers topics on how to extend Orio with new features and components such as program transformation modules.

Writing a New Code Transformation Module

With the module name provided in the leader annotation, Orio dynamically loads the corresponding code transformation module and uses it to transform and generate the code in the annotation body block. If the pertinent module cannot be found in the Orio module directory, an error message is produced, and the Orio optimization process is terminated. Such *name-based dynamic loading* provides flexibility and easy extensibility without requiring detailed knowledge or modification of the existing Orio system.

Since Orio implementation is all in Python, to construct a new program transformation module therefore requires a Python programming skill. Transformation module writers need to create a new Python module inside `src/module` directory, where the abstract base class of the code transformation modules is available. This abstract class provides a partial implementation of the source-to-source transformation process, leaving it to subclasses to complete the implementation. The program below displays the abstract class code that inherits a class constructor and an abstract transformation function to its subclasses. Information about the class attributes is included in the code below as well.

```
#
# File: src/module/module.py
#

class Module:
    '''The abstract class of Orio's code transformation module'''

    def __init__(self, perf_params, module_body_code, annot_body_code, cmd_line_opts, line_no, indent_s
        '''
        The class constructor used to instantiate a program transformation module.

        The following are the class attributes:
          perf_params        a table that maps each performance parameter to its value
          module_body_code   the code inside the module body block
```

```
            annot_body_code    the code contained in the annotation body block
            cmd_line_opts      information about the command line options
                               (see src/main/cmd_line_opts.py for more details)
            line_no            the starting line position of the module code in the source code
            indent_size        an integer representing the number of whitespace characters that
                               preceed the leader annotation
        '''
        self.perf_params = perf_params
        self.module_body_code = module_body_code
        self.annot_body_code = annot_body_code
        self.cmd_line_opts = cmd_line_opts
        self.line_no = line_no
        self.indent_size = indent_size

    def transform(self):
        '''
        The main code transformation procedure. The returned value is a string value that
        represents the transformed/optimized code.
        '''
        raise NotImplementedError('%s: unimplemented abstract function "transform"' %
                                  (self.__class__.__name__))
```

Among all the class attributes, three of them are very important in generating optimized code: *performance parameters*, *module body code*, and *annotation body code*. Performance parameters are information essential for performing code optimization and generation, such as unroll factor and tile size. These parameters are stored in a hashtable to facilitate quick accesses to parameter values. The code contained in the module body block normally outlines the applied optimization techniques and, possibly, the high level description of the computation itself. In order to extract this information, new language syntax and a corresponding parser component must be implemented for each transformation module. In addition, the annotation body code, currently expressed in C language, can electively be parsed and given as input to the transformation module.

In the earlier AXPY-4 case, one could view the module body block as redundant, since the annotation body already contains the same program. The motivation behind requiring Orio users to include the computation itself as part of the annotation is to avoid making use of a full-fledged C compiler infrastructure that potentially will compromise the simplicity, reliability, and portability of Orio. Furthermore, another reason is to allow the computation to be expressed by using domain-specific high-level languages, thus capturing the semantics without imposing tuning constraints resulting from the use of a general-purpose language.

Below, we present a very simple example, which extends Orio with a new module that simply rewrites the annotated code without applying any code transformations at all. The new module has no parser component since there is no necessity to extract information from the annotated code, significantly simplifying the module implementation. First, we need to create a new subdirectory (with a name `simplyrewrite`, for instance) inside `src/module`. And then, we create an *empty* special file `__init__.py` inside directory `src/module/simplyrewrite`, so that Python will know that this folder is a package (i.e. a Python module). After that, a file named `simplyrewrite.py` that contains the implementation of the transformation module class must be defined in directory `src/module/simplyrewrite`. So, here is how the new directory structure will look like after adding the new transformation module extension to Orio.

```
% pwd
/home/username/orio

% tree src/module/
src/module/
|-- __init__.py
|-- module.py
```

```
`-- simplyrewrite
    |-- __init__.py
    `-- simplyrewrite.py

1 directories, 4 files
```

And, below is the detailed code of the `SimplyRewrite` class.

```
#
# File: src/module/simplyrewrite/simplyrewrite.py
#

import module.module

class SimplyRewrite(module.module.Module):
    '''A simple rewriting module'''

    def __init__(self, perf_params, module_body_code, annot_body_code, cmd_line_opts, line_no, indent_s
        '''To instantiate a simple rewriting module'''
        module.module.Module.__init__(self, perf_params, module_body_code, annot_body_code,
                                      cmd_line_opts, line_no, indent_size)

    def transform(self):
        '''To simply rewrite the annotated code'''

        # to create a comment containing information about the class attributes
        comment = '''
        /*
         perf_params = %s
         module_body_code = "%s"
         annot_body_code = "%s"
         line_no = %s
         indent_size = %s
        */
        ''' % (self.perf_params, self.module_body_code, self.annot_body_code, self.line_no, self.indent

        # to rewrite the annotated code, with the class-attribute comment being prepended
        output_code = comment + self.annot_body_code

        # to return the output code
        return output_code
```

In the following, we show the input and output of the newly-added `SimplyRewrite` module.

```
% cat simplyrewrite.py
/*@ begin SimplyRewrite (This is the module body code) @*/
// This is the annotation body code
/*@ end @*/

% orcc -v simplyrewrite.py
====== START ORIO ======

----- begin reading the source file: simplyrewrite.py -----
----- finish reading the source file -----

----- begin parsing annotations -----
----- finish parsing annotations -----

----- begin optimizations -----
----- finish optimizations -----
```

```
----- begin writing the output file(s) -----
--> writing output to: _simplyrewrite.py
----- finish writing the output file(s) -----

====== END ORIO ======

% cat _simplyrewrite.py
/*@ begin SimplyRewrite (This is the module body code) @*/
        /*
         perf_params = {}
         module_body_code = "This is the module body code"
         annot_body_code = "
// This is the annotation body code
"
         line_no = 1
         indent_size = 0
        */
// This is the annotation body code
/*@ end @*/
```

Additional Documentation

- Performance Tuning Specifications

Tools Using Orio

- Pluto -- An automatic parallelizer and locality optimizer for multicores
- PrimeTile -- A parametric multi-level tiler for imperfect loop nests

Papers

- Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium, Rome, Italy, May 25-29, 2009*. To appear. (Preprint ANL/MCS-P1556-1008, bib)

- Boyana Norris, Albert Hartono, and William Gropp. "Annotations for Productivity and Performance Portability," in *Petascale Computing: Algorithms and Applications*. Computational Science. Chapman & Hall / CRC Press, Taylor and Francis Group, 2007. (Preprint ANL/MCS-P1392-0107, bib)